1. **Context**

Après avoir vu le format des données collectées via les API de Binance, dans cette section nous allons nous atteler à stocker ces données dans les bases de données en tenant compte des différentes contraintes.

2. **Choix de la base :**

Il existe plusieurs types de base de données qui peuvent être classées en deux catégories.

- Les bases de données SQL (Structured Query Langage) qui sont des bases de données relationnelles. Ce type de base de données est adapté pour les données extrêmement structurées, organisées en tables reliées entre elles. Ci-dessous des exemples de base de données relationnelles.

    - Oracle
    - PostgreSQL
    - MySQL
    - SQLite

- Les base de données NoSQL (Not Only SQL) sont des bases de données non relationnelles. Ces bases de données ont une nature d'organisation de données sans schéma rendant ainsi plus facilite la gestion et le stockage de vastes volumes de données. Ils peuvent également être facilement mis à l'échelle horizontalement. Ils peuvent être :
    - Des bases de données orientées colonnes
    - Des bases de données orientées documents
    - Des bases de données orientées graphes

  Ci-dessous des exemples de base des données non relationnelles

    - MongoDB
    - Elasticsearch
    - Hbase

Les données historiques collectées via python peuvent être facilement transformées sous formes de dataframe avec des données organisées de façon tabulaire avec des lignes et des colonnes. Nous avons donc décider de choisir une base de données relationnelle (Mysql) pour les données historiques car nous avons à la sortie du collecteur(python) des dataframes structurés facilement injectable dans la base sous forme de table d'une part et d'autre part le langage sql est très populaire et largement utilisé par les Data analyst et Datascientist.

Pour les données en streaming, nous avons un comportement différent par rapport au données historiques. Le volume de données croit continuellement et il n'est pas du tout adapté à une base de données relationnel avec un schéma bien défini. Nous avonsdonc décider de choisir donc une base de NoSQL en occurrence MongoDB.

3. **Implémentation :**

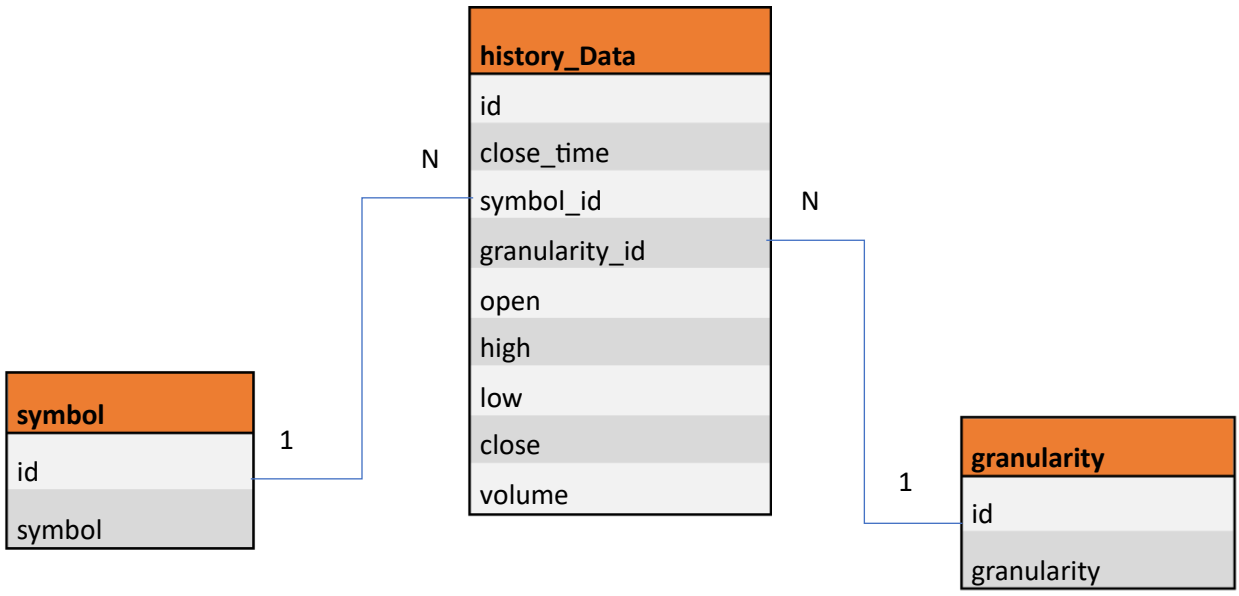
    a) **Mise en place de Mysql**

Pour l'installation de Mysql, nous avons choisi de passer par un docker-composer.yml dont le contenu est décrite ci-dessous :

```yaml
version: '3.3'
services:
  db:
    image: mysql:8.0
    command: --default-authentication-plugin=mysql_native_password
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: $root
      MYSQL_DATABASE: $database
      MYSQL_USER: $user
      MYSQL_PASSWORD: $password
    networks:
      - mysql-compose-network
    ports:
      - "3306:3306"
    expose:
      - '3306'
    volumes :
      # - ./requirements/volumes/data:/volumes/data
      - ./requirements/mysql/volumes/db:/var/lib/mysql
      - ./requirements/mysql/volumes/script:/volumes/script
networks:
    mysql-compose-network:
```

b) Structure de la base de données historiques

Nous avons défini trois tables dans la base de données Mysql suivant le schema relationnel ci-dessous.



La représentation des trois tables :

History_Data(id, close_time, #symbol_id, #granularity_id, open, high, low, close, volume)

symbol (id, symbol)

granularity (id, granularity)

Les différentes tables sont créées à travers un script python dont un extrait est affiché ci-dessous.

```python
import mysql.connector
import os
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

db = mysql.connector.connect(
    host=os.getenv('host'),
    user=os.getenv('user'),
    password=os.getenv('password'),
    database=os.getenv('database')
)

mycursor = db.cursor()

# Creation des tables
mycursor.execute("CREATE TABLE symbol (id INTEGER NOT NULL, symbol
VARCHAR(255),PRIMARY KEY (id))")
mycursor.execute("CREATE TABLE granularity (id INTEGER NOT NULL, granularity
VARCHAR(255),PRIMARY KEY (id))")

# Insertion des valeurs
sql = "INSERT INTO symbol (id, symbol) VALUES (%s, %s)"
val = [
  (1, 'BTCEUR'),
 (2, 'ETHEUR')
]

mycursor.executemany(sql,val)
db.commit()
```

La collecte des données historiques et le stockage dans la base est fait via le script python collect_history_data.py.

Ci-dessous un extract du script de collecte, transformation et d'insertion dans la base des données historiques :

```
# definition de la fonction de collecte
def Recup_data(symbol,id_symbol,id_granularity,decalage):
    date='2021-11-01'
    start_date = datetime.datetime.strptime(date, '%Y-%m-%d')
    #delta = 1000*60*60

    timestamp=int(datetime.datetime.timestamp(pd.to_datetime(start_date))*1000
) # Conversion en timestamp

    if id_granularity == 1:
        klines = client.get_historical_klines(symbol,
client.KLINE_INTERVAL_1HOUR,timestamp)
.
.
.



    data['symbol_id'] = id_symbol
    data['granularity_id'] = id_granularity
    df=data.reindex(columns=['close_time','symbol_id','granularity_id','open',
'high', 'low', 'close', 'volume'])

    return df
……

#Insertion des données dans la base
my_conn =
create_engine("mysql+pymysql://{username}:{pw}@{hostname}/{db}".format(hostnam
e=host_opa, db=database_opa, username=user_opa, pw=password_opa))

train_df.to_sql('history_Data', my_conn, if_exists = 'append', index=False)
```

Une fois les données insérées dans la base, il est facile d'effectuer les requêtes SQL en se connectant directement à la base. Ci-dessous un exemple.

```
mysql> SELECT close_time, symbol_id, granularity_id,close AS close_price FROM
history_Data WHERE symbol_id = 1 LIMIT 10;
+---------------------+-----------+----------------+-------------+
```

```
| close_time          | symbol_id | granularity_id | close_price |
+---------------------+-----------+----------------+-------------+
| 2021-11-01 01:00:00 |         1 |              1 |     53226.6 |
| 2021-11-01 02:00:00 |         1 |              1 |     53458.2 |
| 2021-11-01 03:00:00 |         1 |              1 |     53224.2 |
| 2021-11-01 04:00:00 |         1 |              1 |     52575.4 |
| 2021-11-01 05:00:00 |         1 |              1 |     52042.4 |
| 2021-11-01 06:00:00 |         1 |              1 |     52591.1 |
| 2021-11-01 07:00:00 |         1 |              1 |     52677.4 |
| 2021-11-01 08:00:00 |         1 |              1 |     52778.6 |
| 2021-11-01 09:00:00 |         1 |              1 |       54032 |
| 2021-11-01 10:00:00 |         1 |              1 |     53780.5 |
+---------------------+-----------+----------------+-------------+
```

c) Mise en place de MongoDB

Nous avons choisi également de passer par un fichier docker-compose.yml pour la mise en place de la base MongoDB. Voici un extrait du fichier.

```yaml
version: "3.3"

services:
  database:
    container_name: my_mongo3
    image: mongo:5.0
    environment:
      - MONGO_INITDB_ROOT_USERNAME=$login
      - MONGO_INITDB_DATABASE=auth
      - MONGO_INITDB_ROOT_PASSWORD=$password
    networks:
      - mongo-compose-network
    ports:
      - '27017:27017'
.
.
.
```

d) Insertion des données en streaming

Une fois que les données sont collectées avec python en utilisant le websocket, elles sont insérées dans la base MongoBD sous format json.

Ci-dessous un extract du script de collecte est d'insertion dans la base

```python
def insert_into_mongodb(data):
    data['timestamp'] = datetime.now()
    client = MongoClient(
        host="127.0.0.1",
        port=27017,
```

```
        username=os.getenv('login'),
        password=os.getenv('password')
    )
    db = client.OPA_MongoDB
    collection = db.CandlesticksCollection
    collection.insert_one(data)
.
.
.
ws = websocket.WebSocketApp(socket, on_open=on_open, on_message=on_message,
on_close=on_close, on_error=on_error)

try:
    ws.run_forever()
except KeyboardInterrupt:
    print("WebSocket connection closed by user.")
finally:
    ws.close()
```

Un extrait du résultat d'une requête sur MongoDB est affiché ci-dessous.

```
> db.CandlesticksCollection.find().pretty()
{
        "_id" : ObjectId("65a4f0258b036b979dc8c523"),
        "date" : ISODate("2024-01-15T09:43:17.149Z"),
        "close" : 39101.51,
        "high" : 39101.51,
        "low" : 39101.51,
        "timestamp" : ISODate("2024-01-15T09:43:17.149Z")
}
{
        "_id" : ObjectId("65a4f0268b036b979dc8c525"),
        "date" : ISODate("2024-01-15T09:43:18.231Z"),
        "close" : 39101.51,
        "high" : 39101.51,
        "low" : 39101.51,
        "timestamp" : ISODate("2024-01-15T09:43:18.231Z")
}
```